UNIVERSITY OF EDINBURGH

FACULTY OF SCIENCE AND ENGINEERING

COMPUTER SCIENCE THREE

COMPILING TECHNIQUES

Wednesday 12 June 2002

09:30 to 11:00

Examiners:

Prof. K. Turner (External)

Prof. C. Stirling (Chairman)

INSTRUCTIONS TO CANDIDATES

Answer any TWO questions.

All questions carry equal weight.

1. Consider the following grammar (start symbol S, nonterminals S,A,B,C)

   - $S \rightarrow ABC\$$
   - $B \rightarrow b$
   - $B \rightarrow \epsilon$
   - $C \rightarrow cc$
   - $C \rightarrow \epsilon$
   - $A \rightarrow aAa$
   - $A \rightarrow B$

   (a) Compute FIRST(A) and FOLLOW(A)

   $FIRST(A) = a, b$
   $FOLLOW(A) = a, b, c, \$$

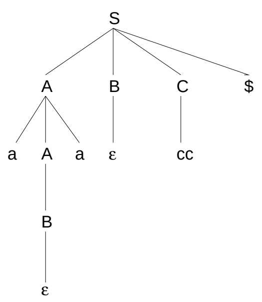
   (b) Write down the entries of the LL(k) parse table for the following pairs of nonterminal/terminal symbols:

   i. A and c: $A \rightarrow B$
   ii. S and \$: $S \rightarrow ABC\$$

   (c) Determine the leftmost derivation to the input “aacc” and write it down as a parse tree.
   Also write down the correspondig parser configurations of a top-down parser given the grammar above and the input “aacc”.

   Derivation $S \xrightarrow{1} ABC\$ \xrightarrow{6} aAaBC\$ \xrightarrow{7} aBaBC\$ \xrightarrow{3} aaBC\$ \xrightarrow{4} aaC\$ \xrightarrow{4} aacc\$$

   Parse Tree:



Goal	Input	Rule
S	aacc$	apply rule 1
ABC$	aacc$	apply rule 6
aAaBC$	aacc$	consume "a"
AaBC$	acc$	apply rule 7
BaBC$	acc$	apply rule 3
aBC$	acc$	consume "a"
BC$	cc$	apply rule 3
C$	cc$	apply rule 4
cc$	cc$	consume "cc"
$	$	consume "$"
ϵ	ϵ	accept

(d) Now consider following grammar (start symbol S, nonterminals S,A,B):

- $S \rightarrow AB\$$
- $A \rightarrow a$
- $B \rightarrow bB|cc$

Write down a recursive-descent parser (in pseudo-notation) for the given grammar. Assume that the lexer is supplied and can be used with the functions below. Omit all file handling. Only specify the functions required for the syntax analysis.

```
const int eof = -1;

// look-ahead token
int tok;

// ``scanner'' returns the next token in
// ``tok'' or eof if the end-of-file is reached
void scanner() { ... }

// Checks if t == tok
// If not:  error!
// Implicitly calls scanner() and supplies next token in tok
void check(int t) { ... }
```

```
main() { scanner(); parser(); }
```

```
parser() { S(); check(eof); }

S() { A(); B(); check($); }

A() { check('a'); }

B() { if (tok == 'b')
      {
         scanner();
         B();
      }
      else {
         check('c');
         check('c');
      }
}
```

Alternative version of B():

```
B() {
  while (tok == 'b')
     scanner();
  check('c');
  check('c');
}
```

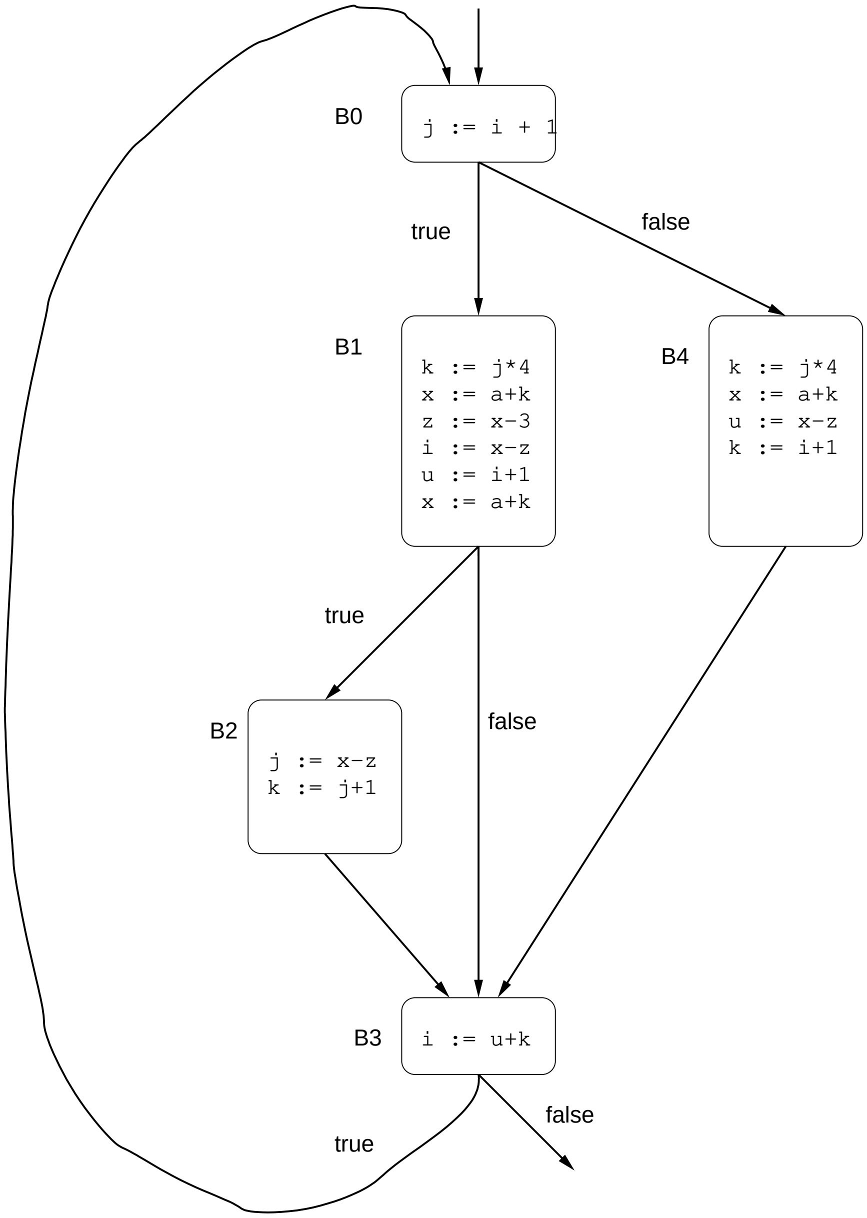
(e) Is it possible for an ambiguous grammar to have the LL(1) property? Justify your answer.

No. For each ambiguous grammar there are several different parse trees for an input. With one symbol look-ahead a top-down parser cannot determine unambiguously which derivation tule to apply. Therefore, any ambiguous grammar cannot be LL(1).

2. (a) Consider the given program fragment. Construct the corresponding Control Flow Graph (CFG) showing all instructions and identify its basic blocks. Label the edges leaving a conditional branch with "true" or "false". Assume that the CJUMP instruction jumps to its first label if the condition evaluates to true, and to the second label otherwise (`CJUMP(condition,`l_{true}`,`l_{false}`)`).
NB: Do not include basic blocks with no computational effect.

```
    def(a)
L1: j := i+1
    CJUMP(<condition>,L2,L3)
L2: k := j*4
    x := a+k
    z := x-3
    i := x-z
    u := i+1
    x := a+k
    CJUMP(<condition>,L6,L7)
L6: j := x-z
    k := j+1
    JUMP(L4)
L7: JUMP(L4)
L3: k := j*4
    x := a+k
    u := x-z
    k := i+1
    JUMP(L4)
L4: i := u+k
    CJUMP(<condition>,L1,L5)
L5: use(a)
    use(i)
```

Draw your Control Flow Graph here:



(b) Assume that there is no aliasing between variables in the program above. Complete the following table, where In[B] and Out[B] represent the variables that are live at, respectively, the entrance to and the exit from a basic block B:

Block B	In[B]	Out[B]
B0	a,i,z	a,i,j,z
B1	a,j	a,k,u,x,z
B2	a,u,x,z	a,u,k,z
B3	a,u,k,z	a,i,z
B4	a,i,j,z	a,u,k,z

Hint: Write data flow on CFG.

(c) Write down a set of covering non-overlapping traces for the CFG constructed in part (a). Try to have as few traces as possible in your covering set.

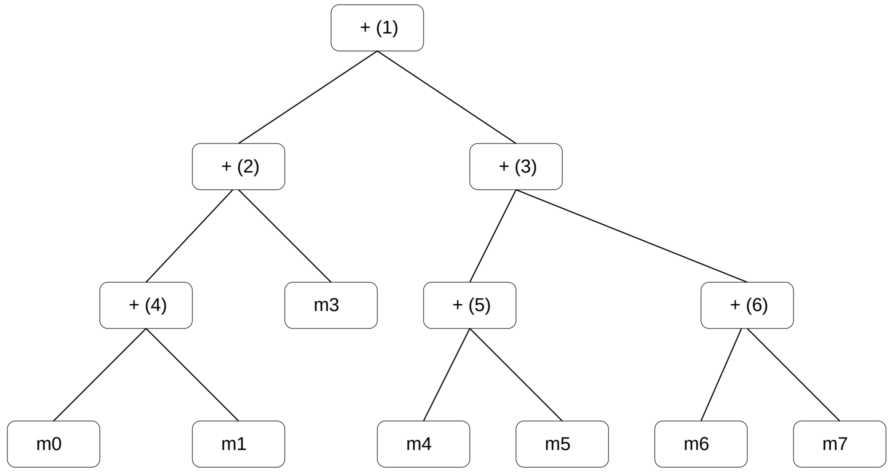
Trace 1: B0, B1, B3, B3

Trace 2: B4

(d) Reorder the basic blocks and alter the CJUMP instructions in such a way that all CJUMPs will be followed by their "false" label. Try to order the blocks so that as many of the unconditional JUMPs as possible are immediately followed by their target block. Write out in full the resulting instruction sequence.

```
LABEL B0:
  j := i+1
CJUMP(condition,B1,B4)
LABEL B4:
  k := j*4
  x := a+k
  u := x-z
  k := i+1

  i := u+k
CJUMP(condition,B0,New)
LABEL New:
JUMP Exit
LABEL B1:
  k := j*4
  x := a+k
  z := x-3
  i := x-z
  u := i+1
  x := a+k
CJUMP(NOT(condition),B3,B2)
LABEL B2:
  j := x-z
  k := j+1
JUMP B3
```

3. (a) Consider the following expression tree (node numbers, given in parentheses next to the internal nodes of the tree, are for identification purposes in the subsequent discussion). Assume that the nodes refer to distinct memory locations and as such need their own registers.



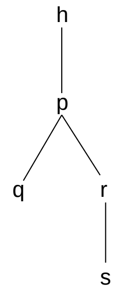
Complete the table below to indicate the Sethi-Ullman number of each node (node numbers refer to the parenthesised numbers in the expression tree above).

Expression Node	Sethi-Ullman Number
a leaf node	1
6	2
5	2
4	2
3	3
2	2
1	3

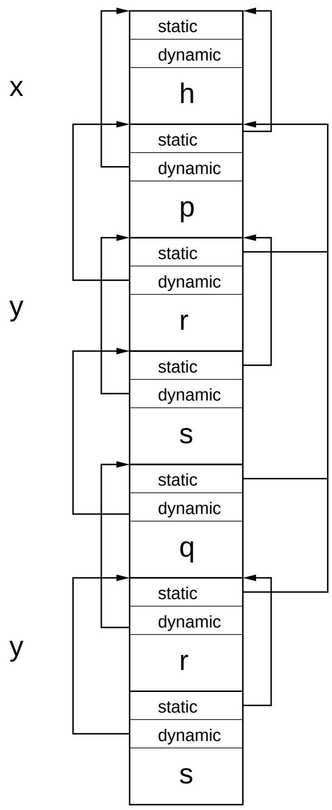
(b) PL/0 allows for nested function definitions. Consider following PL/0 program with its several nested function definitions.

```
PROGRAM h;
   VAR x;

   PROCEDURE p;

      PROCEDURE q;
      BEGIN
         ...
         CALL r;
         ...
      END; // q

      PROCEDURE r;
         VAR y;

         PROCEDURE s;
         BEGIN
           ...
           ... x + y
           CALL q;
           ...
         END; // s

      BEGIN
        ...
        CALL s
        ...
      END; // r

   BEGIN
     ...
     CALL r;
     ...
   END; // p

BEGIN
  ...
  CALL p;
  ...
END. // h
```

i. Sketch the tree of static predecessors for the functions in program h.



ii. Assume the functions of program h have been called in following dynamic order: $h \to p \to r \to s \to q \to r \to s$
Sketch the configuration of the stack with the stack frames stored on it. Include the static and dynamic link fields and show where they point to. Mark the stack frames in which the variables x and y are stored. (Assume that program h is treated as a function, i.e. a separate stack frame will be allocated for the main part of h.)



iii. Variables declared with global scope in program h (such as x) could be stored in a separate stack frame (as suggested in (b)(ii)), but another alternative is to store variables with global scope in an entirely different fixed memory location reserved for global data.

Explain why the variables of global scope can be treated differently from variables declared inside a function. Discuss the advantages and disadvantages of the two options given above when static linkage of the stack frames is used.

- No recursive calls to main program
- No multiple instances of global variables
- No need for stack, variables can be stored in fixed locations

Pro frames: Easier handling, main program can also be handled as function w.r.t. stack frames

Contra frames: Slow access to global variables, static link chain must be traced back

Pro separate data space: Fast access to global variables, no need to trace back static link chain

Contra separate data space: Slightly more complicated, special treatment of global variables necessary